An Interposed 2-Level I/O Scheduling Framework for Performance Virtualization

Jianyong Zhang* Anand Sivasubramaniam* Qian Wang* Erik Riedel† Alma Riska†
*The Pennsylvania State University, University Park
†Seagate Research Center, Pittsburgh, PA
{jzhang,anand}@cse.psu.edu, quw6@psu.edu, {erik.riedel,alma.riska}@seagate.com

Technical Report CSE-05-003
February, 2005

ABSTRACT

I/O consolidation is a growing trend in production environments due to the increasing complexity in tuning and managing storage systems. A consequence of this trend is the need to serve multiple users/workloads simultaneously. It is imperative to make sure that these users are insulated from each other by virtualization in order to meet any service level objective (SLO). Previous proposals for performance virtualization suffer from one or more of the following drawbacks: (i) rely on a fairly detailed performance model of the underlying storage system, (ii) couple rate and latency allocation in a single scheduler making them less flexible, or (iii) may not always exploit the full bandwidth offered by the storage system.

This paper presents a 2-level scheduling framework that can be built on top of an existing storage utility. This framework uses a low-level feedback-driven request scheduler, called AVATAR, that is intended to meet the latency bounds determined by the SLO. The load imposed on AVATAR is regulated by a high-level rate controller, called SARC, to insulate the users from each other. In addition, SARC is work-conserving and tries to fairly distribute any spare bandwidth in the storage system to the different users. This framework naturally decouples rate and latency allocation. Using extensive I/O traces and a detailed storage simulator, we demonstrate that this 2-level framework can simultaneously meet the latency and throughput requirements imposed by an SLO, without requiring extensive knowledge of the underlying storage system.

Keywords

Storage Systems, I/O Scheduling, Quality-of-Service

1. INTRODUCTION

On the hardware end, large disk arrays and networked storage are enabling immense storage capacities and high bandwidth access to this storage in order to facilitate consolidated storage systems. Simultaneously, there is a growing demand on the management and workload side to consolidate storage needs. As storage systems become more complex, they are becoming increasingly difficult to deploy, tune and manage. A substantial investment is required not just to procure such systems, but in the cost of personnel to manage them. It is, thus, more attractive economically to out-source the storage services for consolidation at a data center, whether it be within a single business enterprise, or across enterprises. Further, such consolidation can also facilitate data sharing across these services, which is necessary in some environments, e.g. different applications in the supply-chain of an enterprise may need access to the same data for different purposes. Consequently, we see a growing trend of consolidated data centers, where different workloads/applications/services share the storage infrastructure (sometimes referred to as a *storage utility*) for their end-goals.

While such consolidation is attractive economically, and naturally facilitates data sharing when needed, sharing of the underlying storage infrastructure (including the disk drives, storage caches, network links, switches, controllers, etc.) can lead to interference between the users/workloads/services leading to possible violations in performance-based service level objectives (SLO) that may be binding towards ensuring revenue inflow to the data center. For the purposes of the discussions in this paper, without loss of generality, we assume each user belongs to a different class, with a performance-based SLO agreed upon a priori for each class based on a pricing structure. In order to ensure the revenue stream, the data center would need to insulate the users from each other. Such isolation is referred to as *performance virtualization*, since it gives the impression of the storage utility being fully devoted to each user.

One way of achieving this virtualization is to implement mechanisms within different resources of the storage utility to avoid interference between the classes. For instance, one can implement SLO based disk schedulers (e.g. [10]), interconnect bandwidth allocators (e.g. [28]), cache space managers (e.g. [9, 13]), etc. However, such an option is not very attractive from the practical view-point, since (i) the storage utility can be quite complex making it very difficult to manage each of its resources, (ii) the implementation of the storage utility may not lend itself very well to a good practical model that is usually needed for SLO enforcement, and (iii) the details of the storage utility may not even be available, or its functionality may not be modifiable, in many cases. In a more practical setting, one would like to achieve virtualization by introducing a layer on top of the storage utility which uses very little information about its underlying implementation, i.e., treats the storage utility as a black box. Such an approach, which we follow in our framework, is referred to as an "interposed " [11, 14, 4, 12] scheduler between the user requests and the underlying storage utility as depicted in Figure 1. The interposed scheduler can re-arrange and/or delay requests before dispatching them to the storage utility, but it cannot affect the storage utility subsequently. The interposed scheduler acts as a *QoS gateway* between the stream of client requests and the storage back-end. Real storage systems do sometimes have gateways that manage the traffic between the clients and the storage system, such as, the Logical Volume Manager, the SAN

virtualization switch, etc [4]. Currently, these gateways implement various I/O logic functionalities such as Logical Unit virtualization, online data migration, and remote copy. Our interposed scheduler can reside in these gateways and extend the set of functionalities that these devices offer as depicted in Figure 1.

In our system, incoming I/O requests are classified into different classes, with an SLO pre-determined for each class. Note that latency guarantees and throughput guarantees are equally critical to many (if not all) workloads. Thus, it is important to support both SLO types in a unified framework. While there are some earlier studies [4, 14, 12, 10], which have tried to use such an SLO, most have resorted to an average bound of the latency over a given time period. It is quite possible that a few requests with very low response times can offset the latency violations of a large number of requests. Consequently, our system uses an SLO that is more stringent, requiring *x*% (say 95%) of the requests to have a bounded latency (which is similar with [10]).

In the recent past, there have been some proposals for performance virtualization of storage systems. Some of these proposals (e.g. [10]) use a detailed model of the underlying system to find out how much service is available at any time, and use this information to schedule requests accordingly. However, as we mentioned above, we do not want to use such a model since the utility details may not even be available for the modeling, or even if available the model may not be very accurate. Further, traditional models may not be very suitable for highly transient workload behavior which is becoming a very important optimization criterion these days. Rather than a detailed model, feedback from monitoring the underlying utility can be used to adaptively schedule the requests (as in [14]). Another problem with many of the prior proposals is that they couple together the rate and latency allocation within the design of a single scheduler ([10, 11]). This makes it difficult to find out which goal is more important at any time, to perform the adaptation accordingly. Finally, many prior schedulers are not really work-conserving, i.e. they may not be able to always exploit any spare bandwidth [1] in the underlying storage utility. While one may not want to have a scheduler closely intertwined with the storage utility to gauge any idleness, it is important to be able to at least have some estimate of whether the utility is under-utilized so that we can be "more" work-conserving.

In this paper, we present an interposed 2-level scheduling framework that can meet both latency and throughput guarantees in SLOs for different classes. The higher level mechanism uses a credit-based rate controller (called SARC), to regulate the stream of requests so that they are insulated from each other. In addition, it gets an estimate of whether the utility is being under-utilized from the lower level mechanism (called AVATAR) and tries to distribute this spare bandwidth in a reasonably fair manner across the classes. The low level AVATAR scheduler uses feedback from monitoring the underlying system to regulate requests, i.e., the amount and the order being dispatched to the storage utility. It uses a EDF queue to first order requests based on their latency deadlines, and then based on the relative criticality of response time versus throughput optimization criteria, it dispatches requests from the EDF queue to the storage utility. The mechanisms used in AVATAR are rigorous and based on a careful combination of statistical evaluation of the system and queueing theory results, which increases its flexibility and adaptability and sets it apart from similar approaches [14].

Using synthetic traces and real I/O traces of commercial applications within a detailed storage system simulator, i.e., Disksim [6], we demonstrate that our 2-level framework (SARC+AVATAR) can provide performance isolation and differentiation. It can do a much better job than more recent proposals towards meeting latency and throughput guarantees. It is more adaptive to transient workload behavior, and can quickly compensate to ensure the SLO is being met. Further, it can do a fairly good job of detecting spare bandwidth in the underlying utility, and allocating it in a fair manner to the different classes. This makes it more work-conserving than earlier proposals. Finally, our framework is appealing from the practical viewpoint, is easy to implement, and does not incur high costs.

[1]In this paper, we use the term "spare bandwidth" to refer to the service that is available beyond the current QoS demands in a storage utility.

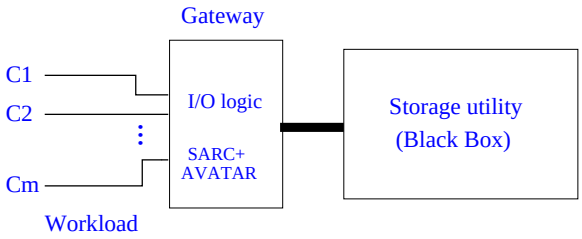


Figure 1: Overall System Model

The rest of this paper is organized as follows. The next section discusses related work. Section 3 gives overview of our framework, together with implementation details of its two components. Section 4 presents results from our evaluation. Finally, section 5 summarizes the contributions of this paper and outlines directions for future work.

2. RELATED WORK

While QoS management in various system resources, particularly for networks [15, 29, 28] and CPU [23, 8] has received a lot of attention, adopting those techniques to I/O is still in its infancy. I/O is quite different from other resources in terms of QoS management [26]. Early work on QoS management for I/O mainly focused on multimedia [7] and real-time workloads [5]. These workloads are very different from the server-based environments that we are targeting, where there is more transient behavior, higher variability across the workloads, and QoS requirements can be quite different.

Storage consolidation, as a way to address the growing complexity and management costs, is increasingly being touted in commercial enterprises. The Storage Network Industry Association (SNIA) has proposed a shared storage architecture and has highlighted the management issues in storage systems [16]. Consequently, more recent work has examined the issue of virtualizing storage, and automating management to a large extent [20]. There are several aspects to storage virtualization [22], including capacity planning, adhering to performance (latency/bandwidth) guarantees, and availability requirements, with different studies focusing on the different aspects.

One way to achieve I/O performance virtualization is to perform resource provisioning/partitioning across workloads in a static manner (e.g. [1, 2]), which is typically coarse-grained. Static (or coarse-grained) resource allocation/partitioning cannot alone handle short-term transient conditions, where much more fine-grained resource management decisions need to be made (usually we may need both kinds of mechanisms in place - resource partitioning at a coarse time granularity, and fine-grained resource control). This paper focuses on the online fine-grained resource control problem.

The main goal of performance virtualization is to meet the throughput and latency requirements (and perhaps fairness issues) of different users/classes, and earlier solutions can be placed in two categories:

1. Schemes in the first category use a proportional bandwidth sharing paradigm. Cello [17] is a two level framework that allocates

bandwidth between classes using an extensive performance model. YFQ [3], SFQ(D), FSFQ(D) [11], and CVC [10] use the Generalized Processor Sharing (*GPS*) principle [15]. These schemes are advantageous in that (i) they provide, at least in theory, a strong degree of fairness (as in GPS), and (ii) they are work-conserving (i.e. they do not let the resource remain idle when there are requests waiting in some class). However, on the downside, the following problems make them less attractive in a practical setting: (i) They need a performance model to estimate the service time of an individual I/O request. While an accurate model for even a single disk drive is not very easy to derive [18, 24], the problem is much more difficult for an array [21]. Further, in the environments that we envision for the applicability of the work, it would be preferable to view the underlying storage device as a black box, and we may not have enough details about the underlying system to get any reasonable performance model. (ii) These schemes couple rate and latency allocation together, making them less flexible, potentially leading to resource over-provisioning [10].

2. Schemes in the second category use feedback-based control to avoid the need of an accurate performance model. Even in this category, schemes fall into two classes:

- Schemes that perform *only* rate control, such as Triage [12] (that does adaptive throttling), SLEDS [4] (that uses a leaky bucket) and RW(D) [11] (that implements a window based rate modulator). These schemes can provide performance isolation and have good scalability. However, they suffer from the following drawbacks: (i) Latency guarantees are not necessarily the intended goals, making it difficult to bound response times; (ii) They are not fully work-conserving [2]. These schemes may not be able to always exploit the full parallelism/bandwidth offered by the underlying storage utility; (iii) When spare bandwidth in the underlying storage utility is observed, these schemes may not be very fair in distributing this spareness to the different classes.
- Schemes that directly deal with the latency guarantee problem, as in Facade [14] which uses combination of real-time scheduling and feedback-based control of the storage device queue. As we show later in the paper, even though this scheme is simple to implement, it cannot easily isolate performance, and is not fast enough in adapting to transient workload changes.

One can envision our framework as combining the benefits of these two solution categories while avoiding their drawbacks, in the following ways:

– Similar to the schemes in the second category, we use a feedback-based mechanism, without requiring an extensive performance model. A real-time scheduler is used in conjunction with such feedback (similar to Facade [14]) in order to provide latency guarantees, unlike the schemes in the first category.

– A high level rate controller is used to regulate the requests from different classes in order to provide performance isolation. This rate controller, coupled with the low level request scheduler, efficiently meets the throughput guarantees across classes which is lacking from schemes such as Facade.

– In addition, similar to the schemes in the first category, our solution allocates spare bandwidth in the underlying storage utility to the different classes, making it more work-conserving than schemes in the second category while still being relatively fair across the classes.

– Unlike the schemes in the first category, our solution decouples rate and latency allocation, making it more flexible and suitable for meeting multi-dimensional requirements.

[2]Note that RW(D) cannot fully utilizes the degree of concurrency offered by the underlying storage utility.

3. THE 2-LEVEL SCHEDULING FRAMEWORK

3.1 Service Level Objectives – SLOs

In our framework, the incoming workload consists of multiple classes, each with a prescribed performance-based Service Level Objective (SLO). The SLO specification for each class i, for $1 \leq i \leq m$ is the tuple $< R_i, D_i >$, where R_i is the maximum class i arrival rate with a latency guarantee requirement of at most D_i. If class i arrival rate is higher than R_i, then its throughput should be at least R_i, but there are no latency guarantee requirements. We call R_i as the rate requirement and D_i as the latency or response time requirement. This SLO specification is enforced in each time interval of predetermined length, i.e., 1 second in our evaluation.

Various schemes that provide latency guarantees [4, 14, 12] maintain the *average* latency within the required SLO bound. However, it is possible that a few fast requests, such as those that represent cache hits offset a large portion of requests that exceed the SLO bound. Consequently in our approach, we use a *statistical latency guarantee*, that is the SLO requires x% (95% in our evaluations), of all requests be bounded by a latency of D_i, for $1 \leq i \leq m$.

3.2 Architecture Principles

Aiming to provide both throughput and latency guarantees, we opt for a 2-level architecture separating rate and latency allocation rather than integrating everything together into a single entity. The higher level of our architecture does traffic regulation via a rate controller, called SARC. The lower level of our architecture provides performance guarantees via a feedback-based controller, called AVATAR. The architecture of our framework is depicted in Figure 2.

The high level rate controller, SARC, is mainly responsible for regulating the incoming traffic to meet the rate requirements and ensuring isolation between classes. SARC aims to fully utilize the underlying storage bandwidth by distributing fairly between all classes any spare bandwidth that is available. SARC manages the incoming traffic in class-based FIFO queues. A credit amount is assigned to each FIFO queue that indicates how many of the outstanding requests can be dispatched to the lower level.

We choose SARC to be a credit-based rate controller because of several advantages. Unlike the leaky bucket mechanism, used in various rate controllers [4], which uses an absolute rate to regulate the traffic, credits capture a relative rate, enabling efficient global coordination across all classes for satisfying SLO requirements. Furthermore, our relative rate representation is easy to be converted into the absolute rate. SARC replenishes class credits not only according to the SLO rate requirements but also every time there is spare bandwidth in the underlying storage utility.

The low level controller, AVATAR, is mainly responsible for satisfying performance guarantees and ensuring effective usage of the storage utility. To meet latency requirements, we use a real-time scheduler, i.e., an EDF queue, which orders all incoming requests based on their deadlines. However, employing just an EDF scheduler may reduce overall throughput, because this queue does not optimize the operation of the storage utility. For example, if the utility is a single disk, one prefers a seek-based or position-based optimization scheme [27] rather than EDF. Nevertheless, it is important to strike a good balance between optimizing for latency and optimizing for throughput. AVATAR uses feedback control to dynamically regulate the number of requests dispatched from the EDF queue to the storage utility, where they may get re-ordered for better efficiency. Note that when dispatching a large number of requests to the storage utility the system is being optimized for efficiency and throughput, while restricting the number of requests at the storage utility gives more priority to deadlines.

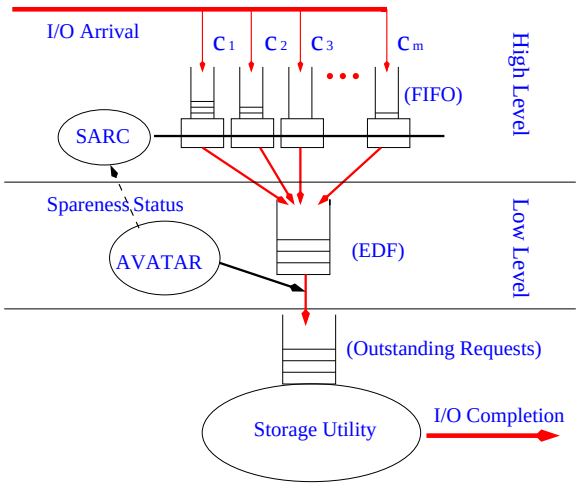


Figure 2: The architecture of our 2-level scheduling framework

We aim to have a work-conserving and fair framework, which requires knowledge of the storage utility utilization status, referred to as the *spareness status*, at the high level rate controller SARC. However, this information is available only at the lower level of our architecture. We use a feedback loop from AVATAR (low-level) to SARC (high-level) which provides to the latter the spareness status of the storage utility.

Upon arrival, an I/O request is tagged with its class id. If the corresponding FIFO queue has any available credit for the request class, then it is dispatched without delay to the low level EDF queue, otherwise it is queued in the FIFO queue waiting until the next replenishment time. In the lower level, AVATAR decides when to dispatch the outstanding requests from the EDF queue to the storage utility. Finally, the storage utility services the outstanding requests based on its own service discipline. Note that our scheme operates outside the storage utility and does not require any changes in it.

3.3 The High Level Rate Controller: SARC

SARC manages the FIFO queues, one for each workload class, and the available credits for each class. Each incoming request consumes only one credit from its class. If one is available, it is dispatched to the lower level; otherwise, it is queued in the corresponding FIFO queue and marked as a backlogged request. SARC has access to the spareness status maintained by AVATAR and described in more detail in Section 3.4.4.

SARC design is related to the amount of class credits and the replenishment policy. Each class i, for $1 \leq i \leq m$, has a maximum amount of credits, denoted by r_i which is directly related to the rate R_i from the class i SLO requirement. Initially, credit amount of class i is set to r_i. During each replenishment event credits of all classes are reset to full amounts regardless of whether or not any class is devoid of credits. A replenishment event happens

- upon a time period, T_{SARC}, having elapsed since the last replenishment event.
- upon a new arrival at a FIFO queue with no available credits, while the spareness status indicates that the storage utility has spare bandwidth available,
- upon AVATAR changing the spareness status and indicating that spare bandwidth has become available,

The rationale for the periodic replenishment event is similar to the goals of a leaky bucket controller [4], in order to ensure that the storage utility bandwidth is distributed across the classes according to their SLO requirements. The second and third replenishment events make sure that there is no backlogged request at the high level if the storage utility has spare bandwidth available. Note that as a consequence of the above replenishment guidelines, no two replenishment events are more than T_{SARC} apart.

Upon a replenishment event (whether it be periodic or due to spareness), each class i, for $1 \leq i \leq m$, is replenished to its full r_i credits, where $r_i = T_{SARC} \cdot R_i$. As can be seen, the credit allocation is determined by the rate agreed upon earlier (SLO) to ensure that not more than r_i requests are allowed for any class during T_{SARC} when the storage system is busy. Since every class can immediately dispatch down to the low level up to $T_{SARC} \cdot R_i$ requests at a replenishment event, the parameter T_{SARC} represents the burstiness of regulated traffic to the lower level.

SARC uses synchronous replenishment, where the credits for all classes are reset at the same time. Note that whenever there are backlogged requests in any class during a replenishment event, they (at most r_i of them) can be dispatched to the low level EDF queue. Intuitively, the synchronous replenishment provides *fairness* in the allocation (determined by the SLO) of any spare bandwidth across the classes.

If the credit replenishments are done only periodically (i.e. remaining oblivious of the storage utility spareness), the credit-based rate controller would not be *work-conserving*, becoming similar to the rate controllers used in SLEDS [4] and Triage [12]. By tracking underlying spareness and distributing that in a relatively fair manner across the classes, SARC becomes more work conserving, i.e., fully utilizes the storage utility in the presence of backlogged requests.

3.4 The Low Level Controller: AVATAR

In the lower level of our architecture, depicted schematically in Figure 3, there are two queueing centers, i.e., the priority EDF queue and the storage utility queue. The flow of requests between these two queues is controlled by AVATAR. While requests are ordered in the EDF queue according to their deadlines (which are set to the sum of the request arrival time and its class latency requirement), their dispatching to the storage utility is controlled by the request class latency requirements and the utilization of the storage utility.

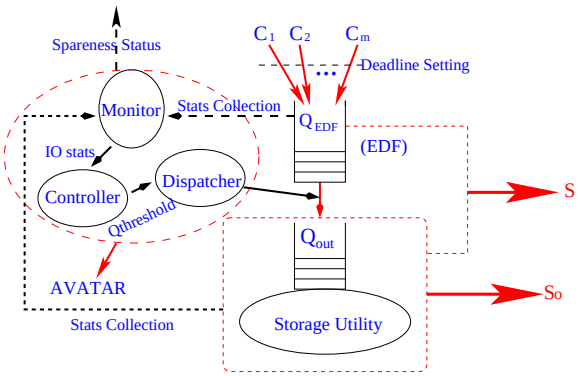


Figure 3: The architecture of the low level scheme

AVATAR is responsible for ensuring that latency requirements of each class are met, making its design and accuracy critically important. Similar to [14], AVATAR uses feedback-based control. However, unlike [14], which is based on various heuristics, AVATAR is based on queueing theory principles and careful approximations. AVATAR combines feedback-based control and Little's law-based bound analysis to periodically set system parameters. We refer to each period, during which AVATAR collects system statistics and

adapts system parameters, a *time window*. The most critical parameter set by AVATAR is the threshold of the storage utility queue length. This threshold is used to control request flow within the low level as well as act like an indicator of the spareness status at the storage utility. As we mentioned in Section 3.2, the spareness status is used by SARC, the high level rate controller of our framework.

AVATAR consists of three main components, namely the *Monitor* which continuously collects IO statistics from the underlying system, the *Controller* which determines the queue threshold at the storage utility for the current time window, and the *Dispatcher* which dispatches the requests from the EDF queue to the storage utility queue guided by the queue threshold set by the Controller. In the following subsections we describe the main components of AVATAR in more detail and explain how AVATAR maintains the spareness status.

3.4.1 AVATAR: the Monitor

The monitor collects various statistics from the underlying storage utility and the EDF queue for any time window k ($k > 0$).

- the number of request arrivals, $L^E_{New}(k)$,

- the number of request arrivals whose deadlines lie in the same time window as well, $L^E_{New-Deadline}(k)$,

- the number of completed requests, $X(k)$;

- average waiting time for requests of class i ($1 \leq i \leq m$) at the EDF queue, $MT^E_i(k)$,

- the 95$^{\text{th}}$ percentile of response time of class i ($1 \leq i \leq m$) at the storage utility (S_o in Figure 3), $T^O_i(k)$,

- maximum number of outstanding requests during the current time window at the storage utility S_o, $L^O_{max}(k)$.

While many of the above statistics are used by the controller as indicators of the system status during the time window k, there are cases when the controller needs to predict the corresponding values for the next time window $(k+1)$. In such cases, we use Last Value Prediction, i.e. estimate the next observation (for time window $k+1$) to be the same as the last measured observation (during the time window k).

3.4.2 AVATAR: the Controller

The critical decision that controller makes, is the periodical update of the queue threshold at the storage utility, denoted by L^O. The threshold is set based on the measured system performance statistics and workload parameters collected by the monitor. The controller makes decisions at the end of each time window k. In the following, we initially give an intuitive explanation of how the queue threshold L^O is controlled by AVATAR and then continue with its algorithmic details.

If there is abundant available bandwidth in the storage utility to accommodate the deadlines of all outstanding requests, we optimize the system for maximal throughput and increase L^O, because the deadlines will be met regardless of the service order at the storage utility queue. Recall that any scheduling optimizations at the storage utility reorders requests for better throughput (e.g., based on their position on the disk). On the other hand, when the system starts missing deadlines, the emphasis is on meeting the deadline and the latency requirements by shortening the storage utility queue, i.e., decreasing L^O , so that requests with incoming deadlines in the EDF queue are given higher priority and dispatched to the storage utility for service. However, if the system gets overloaded, a short queue at the storage utility impacts the storage utility throughput, which causes further cascading of deadline misses. In such extreme cases, the preference is to considerably increase the queue threshold. Actually in overload we choose to set the queue threshold L^O equal to infinity so that the system is optimized for throughput and is able to quickly transition to the under-loaded state.

We assume that the number of requests arriving during any time window at the storage utility approximates the number of requests that leave it. Based on this assumption, we apply Little's law:

$$ML^O = X \cdot MT^O,$$

where ML^O is the mean queue length, X is the throughput, and MT^O is mean response time at the storage utility. Now let's consider Little's law across two successive time windows. We get

$$\frac{ML^O(k+1)}{ML^O(k)} = \frac{X(k+1) \cdot MT^O(k+1)}{X(k) \cdot MT^O(k)} \tag{1}$$

If the system is under steady load, we relate the average queue length and mean response time with the queue threshold and the 95$^{\text{th}}$ percentile of response time via the following approximations:

$$\frac{L^O(k+1)}{L^O(k)} \approx \frac{ML^O(k+1)}{ML^O(k)}, \quad \frac{T^O(k+1)}{T^O(k)} \approx \frac{MT^O(k+1)}{MT^O(k)},$$

where $T^O(k)$ denotes the 95$^{\text{th}}$ percentile of response time at the storage utility during time window k. By combining the above two relations and the Little's law in Eq.(1), we get

$$\frac{L^O(k+1)}{L^O(k)} \approx \frac{X(k+1) \cdot T^O(k+1)}{X(k) \cdot T^O(k)} \tag{2}$$

In a system that operates in steady load, we determine values/ranges for the queue threshold L^O, based on various guidelines related to workload demands and queueing considerations. Below, we explain these guidelines in more details.

Queue Threshold Requirements Based on Response Time Deadlines, L^O_{RT}:

The queue threshold for the next time window $(k+1)$, computed at the end of time window k, is scaled based on how large are the latency requirements relative to the response times. Similar approach is used in [14] as well. If there is a large time gap between a specific request deadline and its completion time, then a large queue threshold L^O is used to utilize and optimize storage utility performance.

In Eq.(2), we assume that the throughput is not significantly different from one time window to the next (i.e. $X(k+1) \approx X(k)$, for $k > 0$), and get

$$\frac{L^O(k+1)}{L^O(k)} \approx \frac{T^O(k+1)}{T^O(k)}.$$

Furthermore, because the storage utility itself does not differentiate requests based on their class, we have

$$\frac{T^O(k+1)}{T^O(k)} \approx \frac{T^O_i(k+1)}{T^O_i(k)}.$$

The class i request deadlines $D^O_i(k)$ at the storage utility S_o are approximated by the difference between their class deadlines at the system S (Figure 3), which includes the EDF queue and the storage utility S_o, and the mean waiting time (MT^E_i) in the EDF queue. The deadline of the class i at the system S is D_i. We use the last value prediction to estimate the mean waiting time of the class i at the EDF queue. Thus we calculate the queue threshold at the storage utility as

$$E_i = \frac{D^O_i(k)}{T^O_i(k)} = \frac{D_i - MT^E_i(k+1)}{T^O_i(k)} \tag{3}$$

$$ {}_iL^O_{RT}(k+1) = E_i \cdot L^O(k) \quad (4)$$

Note that ${}_iL^O_{RT}$ puts a limit on how large the storage utility queue can become before violating the deadlines for a particular class *i*, for $1 \le i \le m$. Consequently, a smaller queue length than the threshold suffices to meet the deadline requirements for that particular class *i*.

Lower Bound on Queue Threshold Based on the Throughput Required to Meet Response Time Demands, $L^O_{\underline{X}}$:

During a time window, at least all requests whose deadlines fall in this time window should be dispatched to the storage utility. This requirement puts a lower bound for the throughput demand that have to meet. The storage utility queue should serve

- requests that are already present in the storage utility at the end of time window *k*, i.e., $L^O_{Exist}(k)$,

- requests in the EDF queue at the end of time window *k*, whose deadlines fall in the next time window $(k+1)$, i.e., $L^E_{Exist-Deadline}(k)$,

- requests that will arrive during time window $(k+1)$ and whose deadlines fall within the same time window, i,e., $L^E_{New-Deadline}(k+1)$. Note that last value prediction is used here.

Assuming that the average response times remain roughly the same from the current time window *k* to the next time window $(k+1)$, (i.e. $MT^O(k+1) \approx MT^O(k)$), we calculate the queue threshold at the storage utility from Eq.(2) for time window $(k+1)$ as:

$$\underline{X}(k+1) = L^O_{Exist}(k) + L^E_{Exist-Deadline}(k) + L^E_{New-Deadline}(k+1) \quad (5)$$

$$L^O_{\underline{X}}(k+1) = \frac{\underline{X}(k+1)}{X(k)} \cdot L^O(k) \quad (6)$$

Upper Bound on Queue Threshold Based on Adequate Throughput, $L^O_{\overline{X}}$:

If the system is operating in underload and the latency requirements are easily met, we find that the queue threshold set using only the response time requirements keeps increasing. At some point, such a phenomenon reduces the role of the EDF queue and leads to deadline misses. Deadline violations subsequently reduce the queue threshold at the storage utility. However, such oscillations in the queue threshold are not preferable and we determine a maximum queue threshold $L^O_{\overline{X}}$ that bounds the queue length at the storage utility when the system tries to optimize only for throughput. For instance, it may suffice for the storage utility to serve all outstanding requests and the newly arrived requests in the EDF queue by the end of a time window *k*, i.e., $L^E_{New}(k)$. Similar to the lower throughput demand, we calculate the upper bound demand as ($L^E_{Exist}(k)$ refers to requests in the EDF queue at the end of time window *k*):

$$\overline{X}(k+1) = L^O_{Exist}(k) + L^E_{Exist}(k) + L^E_{New}(k+1) \quad (7)$$

$$L^O_{\overline{X}}(k+1) = \frac{\overline{X}(k+1)}{X(k)} \cdot L^O(k) \quad (8)$$

Details of the Controller Algorithm

The three statistics that we introduced previously, i.e., L^O_{RT}, $L^O_{\underline{X}}$, and $L^O_{\overline{X}}$, provide the basis for the algorithm of the AVATAR controller. Once these statistics are computed, we evaluate the queue threshold, $L^O_i(k+1)$, for each class *i*, for $1 \le i \le m$. Note that the class specific queue threshold serves as a guideline only for the requests of that specific class. Actually, we use the minimal computed queue threshold across all classes of requests, as to put the most stringent requirements and meet the response time demands.

Our algorithm, shown in Figure 4, categorizes a given time window to be either *overloaded* or *underloaded* for a class of requests. Intuitively, in a period, if the lower bound throughput demand and the response time demand of class *i* can not be satisfied together, the period is overloaded for class *i*; otherwise, it is underloaded for class *i*. If a period is overloaded for all classes, it is in the *overloaded state*; otherwise, it is in the *underloaded state* even though it can be overloaded for some classes. During an overloaded time window, the AVATAR controller sets the queue threshold $L^O_i(k+1)$ to be infinity. The controller knows whether the *previous* time window *k* was underloaded or overloaded for a specific class during its decision making for the next time window $(k+1)$ and uses it as we explain below.

At the end of a time window *k*, the controller first computes the two throughput-based thresholds, $L^O_{\underline{X}}(k+1)$ and $L^O_{\overline{X}}(k+1)$ (line 3-6 in Figure 4), which determine the lower and upper bounds for the queue threshold in the underloaded state. Note that these two bounds are not related to any workload class and generally capture the performance of the storage utility during the current time window *k* and predict load conditions for the next time window $k+1$. Subsequently, AVATAR examines the system state during time window *k* for each class guided by the latency-related threshold ${}_iL^O_{RT}(k+1)$. We explicitly consider the following cases:

- Class *i* was underloaded during time window *k*: For the next time window $k+1$, class *i* remains underloaded again, which requires computation of the queue threshold, or becomes overloaded and the queue threshold is set to infinity. The decision is made after computing the queue threshold requirements based on the response time deadlines for class *i*'s requests for the next window (i.e. ${}_iL^O_{RT}(k+1)$) as shown in Eq.(3) and (4) (line 8-11 in Figure 4). The controller's decision then depends on where the ${}_iL^O_{RT}(k+1)$ falls with respect to the two previously computed throughput-based queue thresholds, i.e., $L^O_{\underline{X}}(k+1)$ and $L^O_{\overline{X}}(k+1)$.

If ${}_iL^O_{RT}(k+1) < L^O_{\underline{X}}(k+1) < L^O_{\overline{X}}(k+1)$ (line 13-14 in Figure 4), then the queue threshold necessary to meet the response time demands falls below the minimum queue length needed to sustain the necessary throughput. This means that setting the queue threshold to ${}_iL^O_{RT}(k+1)$ will reduce the throughput below the required one for meeting the deadlines. This situation indicates overload for class *i* during the next time window $(k+1)$. Hence the target queue threshold for class *i* is set to infinity.

If $L^O_{\underline{X}}(k+1) < L^O_{\overline{X}}(k+1) <_i L^O_{RT}(k+1)$ (line 16-17 in Figure 4), then the system will remain underloaded in the next time window $(k+1)$ as it was during the time window *k*. As such the deadlines will be met easily for class *i* and the queue threshold is set to be $L^O_{\overline{X}}(k+1)$ to optimize storage utility operation for throughput.

If $L^O_{\underline{X}}(k+1) <_i L^O_{RT}(k+1) < L^O_{\overline{X}}(k+1)$ (line 19-23 in Figure 4), class *i* will remain underloaded for the next time window $(k+1)$. However, to set the queue threshold, we need to consider additional information regarding system behavior during the time window *k*. If ${}_iL^O_{RT}(k+1) < L^O(k)$, then the requirements for the next time window $(k+1)$ are more stringent than during the time window *k*. So the queue threshold is reduced from $L^O(k)$ to ${}_iL^O_{RT}(k+1)$ for the next time window $(k+1)$ (line 19-20 in Figure 4). Otherwise, if the arrival traffics demand a larger queue length ($L^O_{max}(k) \ge L^O(k)$), we increase the queue threshold to ${}_iL^O_{RT}(k+1)$ (line 19-20 in Figure 4). If ${}_iL^O_{RT}(k+1) \ge L^O(k)$ and $L^O_{max}(k) < L^O(k)$, then the storage utility throughput during the time window *k* will suffice for the next time window $(k+1)$ and the response time requirement also can be satisfied, thus this is a balance state. We only need to carry the same value $L^O(k)$ as the queue threshold for the time window $(k+1)$ (line 22-23 in Figure 4).

- **Class i was overloaded during the time window k:** Because class i is already overloaded with queue threshold infinity, we guide our decision using the actual maximum queue length at the storage utility for the time window k and calculate ${}_iL^O_{RT}$ for the time window $(k+1)$ (line 25-26 in Figure 4). For the time window $(k+1)$, a class that was overloaded during the time window k becomes either underloaded or remains overloaded again. If the required throughput requirements are not met during the time window k, we continue to keep an infinite queue threshold for class i, because it continues to be overloaded (line 31-32 in Figure 4).

The transition from an overloaded to an underloaded state is detected by checking if the minimum number of requests to be serviced during the next time window $(k+1)$ is lower than the number of requests served during the time window k. We use only 90% of this value to avoid possible oscillations. When the system is transitioning to a less loaded state, we bring down the queue threshold (which was infinity) and set it to the maximum of the queue length required to meet the response time deadlines or to maintain the minimal throughput (line 28-29 in Figure 4).

Note that all the above choices are for calculating the queue threshold at the storage utility from the perspective of a single class. In order to eventually set the overall storage utility queue threshold, we set it to be the minimum across all m classes that are being served by the storage utility (line 34 in Figure 4).

3.4.3 AVATAR: the Dispatcher

The role of the AVATAR dispatcher is to dispatch requests from the EDF queue to the underlying storage utility. By default any request in the EDF queue that missed its deadline is dispatched to the storage utility queue regardless of the queue threshold of the latter. Further, the dispatcher selects as many requests from the EDF queue as necessary to bring up the number of the outstanding requests at the storage utility to the queue threshold at any of the following times: (i) when new requests arrive at the EDF queue; (ii) when requests depart from the storage utility upon completion of service; (iii) at the beginning of each time window.

3.4.4 Spareness detection

As mentioned in Section 3, it is of critical importance for AVATAR to maintain the spareness status, so that our approach remains work-conserving. AVATAR sets the queue threshold so that the storage utility is fully utilized and the SLO requirements are not violated. We consider the queue threshold L^O as the degree of concurrency at the storage utility. Thus if the number of actual outstanding requests is less than the degree of concurrency, we consider the storage utility to have spare bandwidth. Because AVATAR characterizes the storage utility state as either underloaded or overloaded, we consider spareness detection in both of these two system states.

If the system is in the underloaded state, we compare the current number of outstanding requests (L_{curr}) and the queue threshold (L^O). If $L_{curr} < L^O \cdot 0.9$, we consider the storage utility to have spare bandwidth. We use 90% of the queue threshold for stability reasons and avoid prediction errors that would lead to assuming there is spare bandwidth when there is none available. If the system is in the overloaded state, then it is clear that the storage utility has no spare bandwidth. Since the spareness status can change only (i) when requests are dispatched to the lower level; (ii) when requests depart from the storage utility; (iii) at the begining of an overloaded time window, we update the spareness status using the above guidelines only at these points in time.

3.5 Properties of the Proposed Framework

In this subsection, we discuss how the proposed framework can successfully guarantee critical properties such as work-conservation, fairness, performance guarantees, and isolation.

AVATAR$(k+1)$

1 $/*$ Called at the end of time window $k*/$
2 $/*$ Set threshold $L^O(k+1)$ for time window $k+1*/$
3 $\underline{X}(k+1) \leftarrow L^O_{Exist}(k) + L^E_{Exist-Deadline}(k) + L^E_{New-Deadline}(k+1)$
4 $\overline{X}(k+1) \leftarrow L^O_{Exist}(k) + L^E_{Exist}(k) + L^E_{New}(k+1)$
5 $L^O_{\underline{X}}(k+1) \leftarrow L^O(k) \cdot \underline{X}(k+1)/X(k)$
6 $L^O_{\overline{X}}(k+1) \leftarrow L^O(k) \cdot \overline{X}(k+1)/X(k)$
7 **for** class $i \leftarrow 1$ **to** m
8 **do** $E_i \leftarrow (D_i - MT^E_i(k+1))/T^O_i(k)$
9 **if** $L^O_i(k) < \infty$
10 **then** $/*$ underload case $*/$
11 ${}_iL^O_{RT}(k+1) \leftarrow E_i \cdot L^O(k)$
12 **switch**
13 **case** ${}_iL^O_{RT}(k+1) < L^O_{\underline{X}}(k+1)$:
14 $L^O_i(k+1) \leftarrow \infty$
15
16 **case** ${}_iL^O_{RT}(k+1) > L^O_{\overline{X}}(k+1)$:
17 $L^O_i(k+1) \leftarrow L^O_{\overline{X}}(k+1)$
18
19 **case** ${}_iL^O_{RT}(k+1) < L^O(k)$ *or* $L^O_{max}(k) \geq L^O(k)$:
20 $L^O_i(k+1) \leftarrow_i L^O_{RT}(k+1)$
21
22 **case** ${}_iL^O_{RT}(k+1) \geq L^O(k)$ *and* $L^O_{max}(k) < L^O(k)$:
23 $L^O_i(k+1) \leftarrow L^O(k)$
24
25 **else** $/*$ overload case $*/$
26 ${}_iL^O_{RT}(k+1) \leftarrow E_i \cdot L^O_{max}(k)$
27 **switch**
28 **case** $\underline{X}(k+1) \leq X(k) \cdot 0.9$:
29 $L^O_i(k+1) \leftarrow max({}_iL^O_{RT}(k+1), L^O_{\underline{X}}(k+1))$
30
31 **case** $\overline{X}(k+1) > X(k) \cdot 0.9$:
32 $L^O_i(k+1) \leftarrow \infty$
33
34 $L^O(k+1) \leftarrow \min_i L^O_i(k+1)$
35 **return** $L^O(k+1)$

Figure 4: Pseudo Code of the AVATAR Controller

3.5.1 Work-conservation and Fairness

In our framework, AVATAR continuously detects the spare bandwidth of the underlying system and SARC uses it to serve the backlogged requests aggressively. Our scheme makes the storage utility more work-conserving through utilizing as much of its concurrency as possible. Its efficiency depends on the accuracy of spareness detection by AVATAR.

A good strategy to use the spare bandwidth is to fairly distribute it across all classes. A fair scheme should distribute spare bandwidth according to a well-defined policy and does not penalize a class for using excess bandwidth. In our scheme, we use the rate-proportional fairness approach, where the spare bandwidth is distributed across all classes proportional to their rate requirements specified in their SLOs. Intuitively, our scheme always distributes the entire bandwidth among classes via the credit replenishment mechanism and each backlogged class consumes the spare bandwidth proportional to its full credits. If our scheme detects that there is spare bandwidth in the storage utility, it replenishes credits of all classes, so as to not penalize any class that uses excess

bandwidth. In the following, we present a formal analysis for the evaluation of the fairness of our policy.

First, we analyze classes that are backlogged in the high level FIFO queues of our framework. If a class i, for $1 \leq i \leq m$, continuously has outstanding requests in its FIFO queue during an interval $[T_1, T_2]$, then we consider class i to be backlogged in interval $[T_1, T_2]$.

THEOREM 1. During any time interval $[T_1, T_2]$, the difference between the number of requests released by SARC for two backlogged classes i and j, for $1 \leq i, j \leq m$, is given as

$$\left| \frac{W_i(T_1,T_2)}{R_i} - \frac{W_j(T_1,T_2)}{R_j} \right| \leq T_{SARC},$$

where $W_l(T_1,T_2)$ for $l = i, j$ is the number of class l released requests from the high level FIFO queue during $[T_1, T_2]$, R_l is the class l rate requirement specified in its SLO ($< R_l, D_l >$), and T_{SARC} is the parameter used in SARC.

[Proof]: If class i or j, for $1 \leq i, j \leq m$, is backlogged during this interval, its credits are used up immediately after being replenished. Since no other request in these classes will be dispatched until the next replenishment, SARC only dispatches the requests for the backlogged classes at replenishment time.

Based on this observation, if there is no replenishment event in the interval $[T_1, T_2]$, then $W_i(T_1,T_2) = W_j(T_1,T_2) = 0$. Otherwise, let $S = \{t_1, \ldots, t_n\}$ be the sequence of time-stamps of replenishment events during the interval $[T_1, T_2]$. At each of these times, $t_1, \ldots, t_{n-1}$, the number of requests released into the system for both classes i and j is equal to their full credits (denoted as r_i or r_j, respectively for class i or j). Further, for t_n, the number of requests released for classes i and j would be at most r_i and r_j, since (i) if $t_n = T_2$, thus t_n is the end of the current backlogged interval, then the number of released requests is smaller value between the number of backlogged requests in T_2 and the full credits, (ii) otherwise, the number of released requests would be equal to the full class credits. Consequently, during time interval $[T_1, T_2]$,

$$r_i \cdot (n-1) < W_i(T_1,T_2) \leq r_i \cdot n \;\Rightarrow\; n-1 < \frac{W_i(T_1,T_2)}{r_i} \leq n,$$

$$r_j \cdot (n-1) < W_j(T_1,T_2) \leq r_j \cdot n \;\Rightarrow\; n-1 < \frac{W_j(T_1,T_2)}{r_j} \leq n.$$

and

$$\left| \frac{W_i(T_1,T_2)}{r_i} - \frac{W_j(T_1,T_2)}{r_j} \right| \leq 1.$$

Since $r_i = R_i \cdot T_{SARC}$ and $r_j = R_j \cdot T_{SARC}$, then

$$\left| \frac{W_i(T_1,T_2)}{R_i} - \frac{W_j(T_1,T_2)}{R_j} \right| \leq T_{SARC}$$

[]

Theorem 1 proves that request dispatching from SARC follows rate-proportional fairness for the number of dispatched requests. We consider T_{SARC} to be a dispatching fairness indicator. In our experiments, we find that our framework performs well for $T_{SARC} = 0.2sec$. Large values of T_{SARC} will cause the framework to behave less fairly and more bursty than what what we present in Section 4. Upon being dispatched to the low level, all requests are controlled by AVATAR, so that they are served based on their latency requirements.

3.5.2 Performance guarantees and isolation

By using a 2-level mechanism, our scheme naturally decouples rate and latency allocation. For any class i with SLO $< R_i, D_i >$ for $1 \leq i \leq m$, we determine the replenishment credits r_i based on the class's rate requirement R_i and the deadline of each request based on its class's latency requirement D_i. However, for schemes that couple rate and latency allocation together, such as CVC [10], SFQ(D), FSFQ(D) [11], the tuple of the SLO requirements $< R_i, D_i >$ is combined into only one parameter (i.e., the weight of the class i). By restricting themselves to one parameter (instead of two) to express the SLO, these schemes are restricted in their ability to adapt to dynamic system conditions and may result in resource over-provisioning for specific classes.

Recall that our framework operates in fine-grained time scales, i.e., seconds and minutes, and as such should be complemented by a QoS-aware provisioning tool such as Hippodrome [2]. These tools operate in coarse-grained time scales such as hours and days and provision enough resources to satisfy the SLOs, avoiding over-provisioning for cost reasons. They set the system in a deterministic state of resource levels during fine-grained time scales and leave it unprotected from transient overloads. Our high level controller, SARC, regulates traffic, ensures work-conservation, and fairness during fine-grained time scales. The low level controller AVATAR, on the other hand, adaptively detects underload and overload states. AVATAR provides latency guarantees during underload and graceful degradation during transient overloads reducing their impact on performance guarantees. Note that the latency guarantees by the low level scheme do not take into account the time spent at the high level FIFO queues. Hence, AVATAR guarantees latency requirements if a class arrival rate is less than the rate requirement in its SLO and its requests are dispatched immediately to the low level EDF queue. If the traffic for a specific class exceeds its SLO rate requirement, then all additional requests, are queued at the high level FIFO queue. Consequently there is no latency guarantee for this specific class, but there are throughput guarantees because the high level ensures that the additional requests are dispatched based on the class SLO rate requirement and the low level schedules them timely at the storage utility.

Our framework distributes dynamically the available bandwidth across classes according to their SLO requirements. If there is spare bandwidth, SARC replenishes the credits of *all* classes. Consequently, a sudden load burst in one class benefits from the spare bandwidth in the storage utility without affecting the flow of requests from other classes, because they have their credits replenished as well. Thus, our framework ensures performance isolation. In fact, fairness in our framework provides strong performance isolation between classes.

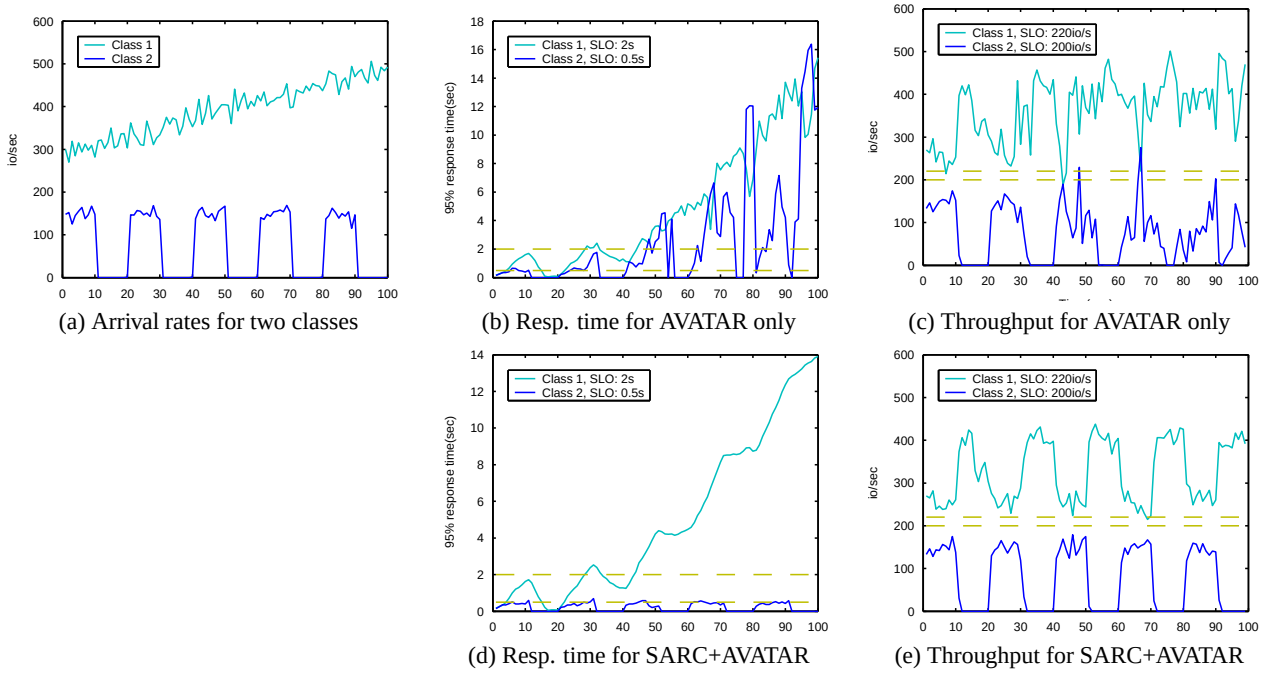
4. EXPERIMENTAL EVALUATION

We evaluate our 2-level framework via simulation-based analysis driven by both synthetic and real workloads. For all synthetic workloads, we use Poisson arrivals with a read-write ratio of 2:1. We use Disksim 2.0 [6] as the detailed storage system simulator. The simulated underlying storage utility is a RAID 5 system with 8 Seagate Cheetah9LP 10K RPM disks. The RAID system uses a SCSI interconnect with 80MB/s transfer rate. The array controller has 128 MB cache and each individual disk has 1 MB of cache. The write policy is "write-through combining with immediate report" (i.e., each write request is reported to be complete as soon as it is cached and the data is sent immediately to the disk).

We designed and conducted various experiments that illustrate the performance guarantees of our framework. In all our experiments, we assume that the system operates under a two-class workload.

4.1 E1: Performance Isolation With SARC

Our first experiment (E1) focuses on performance isolation in overload conditions. Since isolation is achieved by the high-level rate



(a) Arrival rates for two classes (b) Resp. time for AVATAR only (c) Throughput for AVATAR only

(d) Resp. time for SARC+AVATAR (e) Throughput for SARC+AVATAR

Figure 5: Illustrating performance isolation abilities of SARC (E1). The x-axis is time (sec).

controller (SARC), to illustrate its effectiveness and need, we compare the results for our 2-level framework (SARC+AVATAR), with one which uses only AVATAR to meet latency requirements.

The trace that drives the simulation in this experiment is 100 seconds long, as shown in Figure 5. Class one in the trace starts with a $300io/s$ Poisson arrival rate and increases it by $20io/s$ every 10 seconds, reaching $480io/s$ by the end of the trace. Class two in the trace represents an ON/OFF workload that is ON for 10 seconds with the Poisson arrival rate of $150io/s$ and OFF for 10 seconds. The SLO for classes one and two are $< 220io/s, 2000ms >$ and $< 200io/s, 500ms >$ respectively. The aggregate rate of $450io/s$ saturates the system and the storage utility operates in overload for all ON periods of class two except the first one.

Response time and throughput over the duration of the experiment are presented in Figures 5(b) and (c), respectively, for a system using only AVATAR. In Figures 5(d) and (e), we present the same results for the SARC+AVATAR scheme. Since class one always exceeds the SLO rate requirement of $220io/s$, the scheme does not ensure latency guarantees for this class, but should provide a minimum throughput of $220io/s$. Class two does not exceed its SLO rate requirement and the scheme should ensure latency of at most $500ms$ for this class.

Observe that SARC+AVATAR meets the SLOs requirements for both classes, while AVATAR only, lacking the rate controller, does not effectively isolate the two classes. Figure 5(b) indicates that soon after approximately 20 seconds, the latency requirement for class two are violated, and the response times keep increasing during the ON periods. Figure 5(c) shows that many requests of class two are being serviced well into the middle of the OFF period for the AVATAR only execution. If the throughput requirements of class two are indeed being satisfied, then its requests should have completed at the end of the ON period, or at most one second beyond that point (as in the case of Figure 5(e)) since there are no new requests in the OFF period. Hence, the AVATAR only scheme does not meet the throughput requirement either. Figures 5(d) and (e) show the benefit of the rate controller SARC and its efficiency to achieve performance isolation.

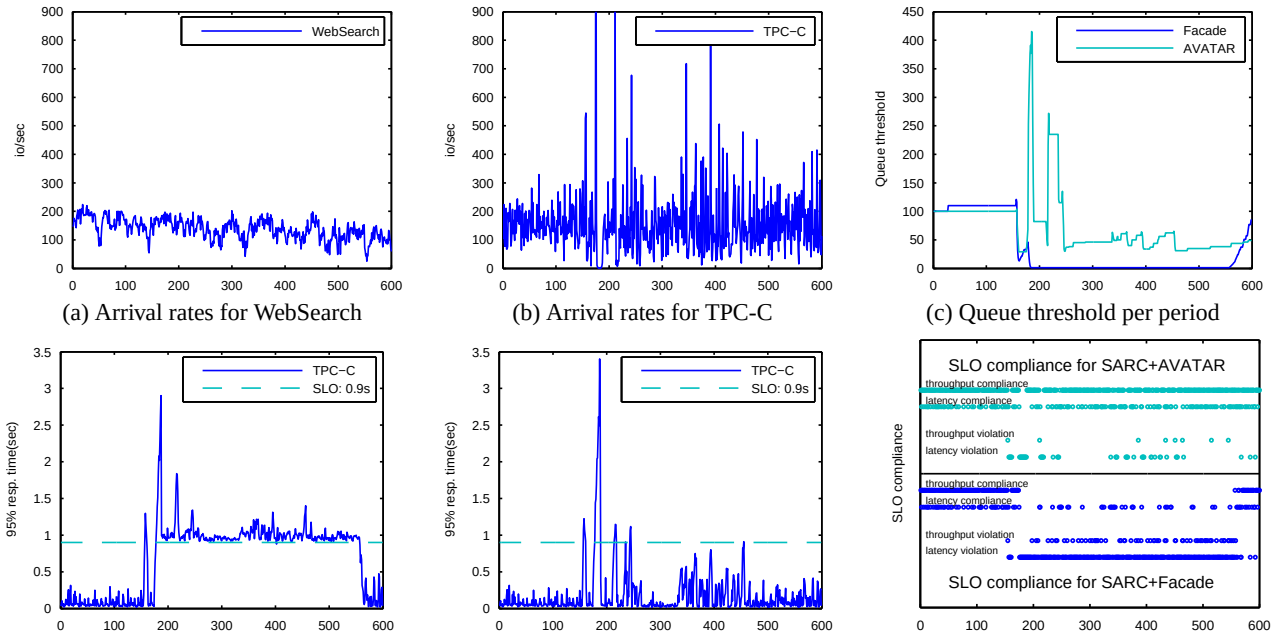
Observe in Figure 5(e) that the class one throughput under SARC+AVATAR follows a periodic sinusoidal behavior, reaching a peak of approximately $450io/s$, i.e., the maximum system capacity, which shows the effectiveness of our framework in detecting spare bandwidth (during the class two OFF periods) and allocating it to the class one. This is further explored in Section 4.3.

4.2 E2: Adaptability of AVATAR

Having shown the effectiveness of the high level rate controller SARC in providing isolation, we illustrate the adaptability of AVATAR in meeting the throughput and latency requirements. We compare AVATAR with the only other scheme that uses feedback-based control and similar SLO representation, Facade [14]. We provide the high level rate controller SARC to both schemes, i.e., AVATAR and Facade, as to evaluate a system with the same level of performance isolation.

In this experiment, the simulation is driven by workload that is the mix of two I/O traces measured in real systems and run for 600 seconds. The first trace [25] comes from a system that deploys a Web Search Engine. The second trace [30] comes from an OLTP-type of system set up according to the TPC-C specifications, where the entire database size is approximately 25 GB. The arrival process characteristics of these two traces are shown in Figure 6(a) and (b), respectively. We set the SLOs to be $(160io/s, 400ms)$ for the WebSearch class (class one) and $(190io/s, 900ms)$ for the TPC-C class (class two). Figure 6(b) shows that the TPC-C class is bursty. There are times when its arrival rates are above $1000io/s$ (truncated in Figure 6(b)) resulting in a transient overload at the storage utility.

In Figures 6 (c), (d), and (e), we present the experimental results for the TPC-C class only. The results for the WebSearch class are similar and we omit them here for lack of space. Figures 6(d) and (e) show the 95^{th} percentile of request response times at the low level. Note that in our framework, if the request deadlines of a class are satisfied at the low level, then both throughput and latency require-



(a) Arrival rates for WebSearch (b) Arrival rates for TPC-C (c) Queue threshold per period

(d) Resp. time of the low level for Facade (e) Resp. time of the low level for AVATAR (f) SLO compliance for TPC-C

Figure 6: Illustrating the adaptability of AVATAR (E2). The x-axis is time (sec).

ments for that class are satisfied overall. Thus we use the response times at the low level as an indicator of performance guarantees. In order to illustrate the bigger picture of performance guarantees, Figure 6 (f) shows the SLO compliance of the two schemes in terms of whether the scheme meets the throughput requirements when the arrival rate is higher than that specified in the SLO, whether it meets the latency bound when the arrival rate is lower, or it does not meet the appropriate SLO.

Figure 6 (f) shows that SARC+AVATAR has a much better SLO compliance than SARC+Facade. For instance, there are many more periods of latency violation (and also throughput violations) for SARC+Facade than SARC+AVATAR. Figures 6 (d) and (e) give insights on this behavior. SARC+AVATAR satisfies all request deadlines except for a few overloaded intervals (Overall the miss ratio for SARC+AVATAR is less than 5%). Observe that during the time interval of $(180, 580)$ seconds, the response time under (SARC+AVATAR) is less than the SLO latency requirement most of the time, when under SARC+Facade the request deadlines are continuously missed (Overall the miss ratio for SARC+Facade is above 50%). The reasons behind this difference in performance lies in the different ways that Facade and AVATAR handle overload.

According to the storage utility queue lengths, presented in Figure 6(c), the overload starts approximately at the 176^{th} second. Facade detects overload by monitoring *only* the arrival rate and comparing it with the SLO rate requirement. Because SARC regulates the rate of requests dispatched to the low level, Facade does not consider the system to be overloaded at this time. However, since the system starts missing deadlines, Facade aggressively reduces the queue threshold, L^O, at the storage utility, which reaches as low as 1 (see Figure 6(c)) at approximately the 180^{th} second. This low value remains until approximately the 580^{th} second, even though the load is not always high during the entire period. The correct action during the transient overload is to increase the threshold considerably so that the storage utility is optimized for throughput rather than latency. AVATAR uses current system conditions rather than arrival rate to immediately detect, handle, and recover from the transient overload. Observe in Figure 6(c) that AVATAR is able to set the queue threshold high and adapt to transient overload.

In Table 1, we present the SLO violation ratios (fraction of periods where the scheme failed to meet the SLO) and average response times for both classes for the entire experiment. They support the above time-varying observations, that SARC+AVATAR provides better performance than SARC+FACADE.

	WebSearch		TPC-C	
	SLO viol. ratio	Avg. rt	SLO viol. ratio	Avg. rt
Facade	64.2%	342.05	60.8%	626.68
AVATAR	4.3%	58.50	10.5%	97.42

Table 1: Results for E2. Response times are in msecs.

Experiment 2 emphasizes the importance of detecting overload in a timely manner, optimizing the system performance for throughput, and recovering from the transient overload condition. The adaptability in AVATAR is related to the wide range of system statistics and metrics that it uses to make decisions, as explained in the previous section, which is missing in Facade.

Note that one can appropriately construct an SLO to accommodate the overloaded conditions. For instance, if the TPC-C workload used an SLO of $(180io/s, 900ms)$ instead of $(190io/s, 900ms)$, then SARC+FACADE would perform as well as SARC+AVATAR. However, such a mechanism requires (i) extensively tuning and refinement of the SLO (a priori), which is not straightforward, and (ii) it results in resource over-provisioning during underloaded, i.e., normal conditions, due to the conservative SLO.

4.3 E3: Spare Bandwidth Utilization

Experiment E3 is designed to evaluate the effectiveness of our high level rate controller SARC in utilizing spare bandwidth. We compare its performance with two other rate controllers that do not use spareness detection. The first approach, called FIXED, periodically

replenishes full credits for each class, with no other replenishments otherwise. FIXED is in the same spirit as the leaky bucket mechanism with fixed parameters. The second scheme, called ADAPTIVE, sets the full credit amounts for each class at 1 second intervals. ADAPTIVE is similar to FIXED except that it changes the full credit amounts for classes every second adaptively based on the SLO requirements and load conditions. ADAPTIVE is in the same spirit as SLEDS [4]. See Appendix A for more details on the ADAPTIVE approach. Note that for the ADAPTIVE scheme, we tried various values for its critical parameters, i.e., P_{rt}, P_{inc}, and P_{dec} (see Appendix A), and select the set that generates the best results in each experiment. All three rate controllers use AVATAR as the low level scheduler.

We evaluate the three schemes using initially synthetic I/O traces and then real workloads. The synthetic trace is 100 seconds long and is shown in Figure 7 (a). Class one in the trace has constant Poisson arrival rate of $400io/s$ for its entire duration. Class two in the trace represents an ON/OFF workload that is ON for 10 seconds with a Poisson arrival rate of $300io/s$ and OFF for 10 seconds. An aggregate rate of about $400io/s$ saturates the system. As such the storage utility is always in overload. The SLO for classes one and two are, respectively, $< 160io/s, 2000ms >$ and $< 160io/s, 500ms >$. The SLO requirements are set loose (and satisfied by all schemes) so that the storage utility has spare bandwidth to be effectively used by the three rate controllers that we evaluate. In this experiment, we set the parameters of ADAPTIVE as $P_{rt} = 40\%$, $P_{inc} = 4\%$, and $P_{dec} = 4\%$.

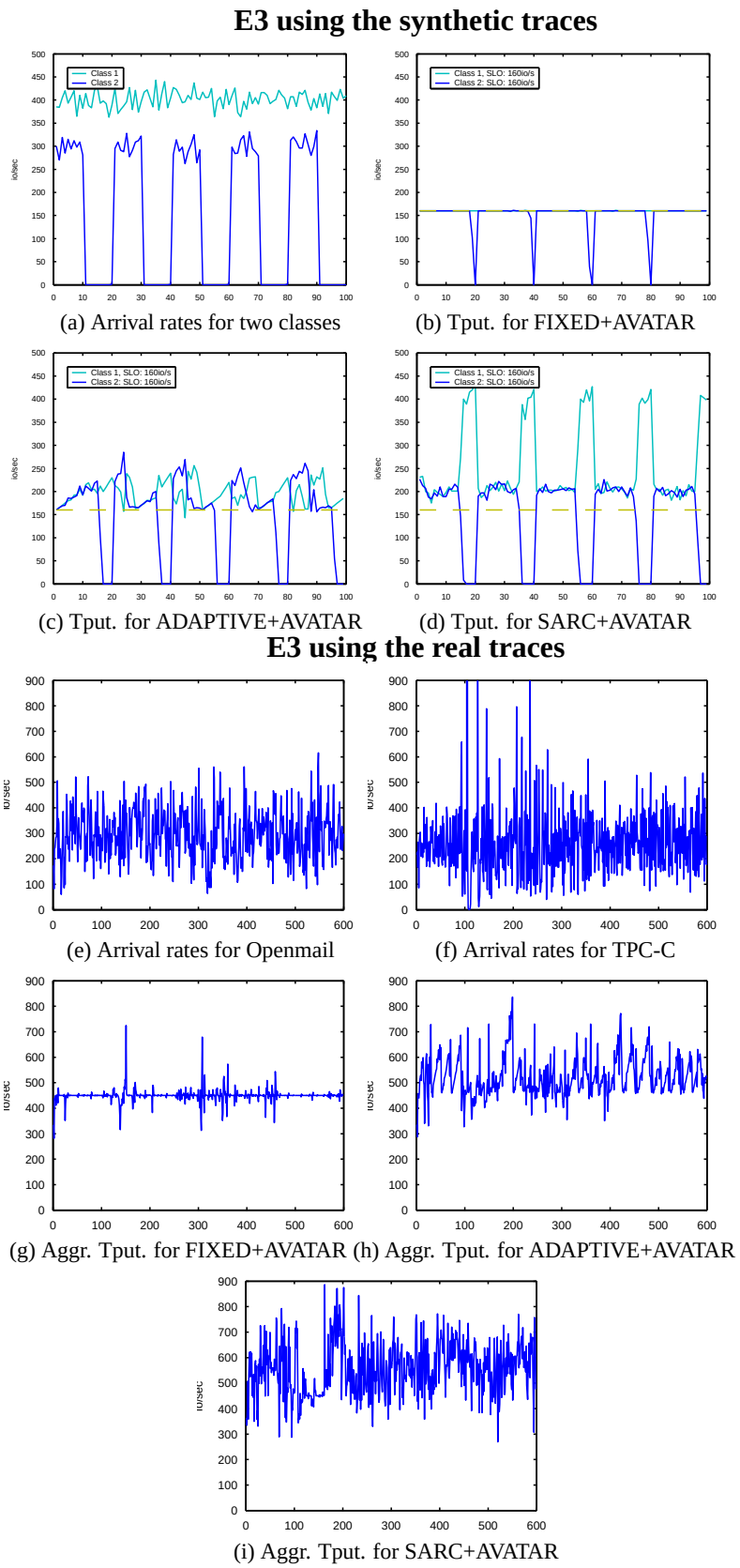
Figure 7(b), (c), and (d) show the throughput of both classes for FIXED, ADAPTIVE, and SARC, respectively. FIXED+AVATAR (Figure 7(b)) limits the dispatching rate to $160io/s$ as specified by the SLO, not using any spare bandwidth. ADAPTIVE+AVATAR (Figure 7(c)) dispatches at a rate higher than $160io/s$ for both classes but does not use the full capacity of $400io/s$ in the storage utility. Note that during the OFF periods of class one ADAPTIVE+AVATAR reaches a throughput of $250io/s$ for class two. Observe that SARC+AVATAR (Figure 7(d)) is able to effectively use the spare bandwidth for the duration of the experiment, pushing the aggregate throughput to approximately $400io/s$. We stress that the results in Figure 7(d) indicate both SARC's effectiveness and fairness in using any spare bandwidth in the system. Observe that during the class one ON periods, SARC achieves comparable throughput for both classes (achieving fairness), while during the class one OFF periods almost all the available bandwidth is used by class two (exploiting spare bandwidth).

	FIXED	ADAPTIVE	SARC
Openmail	249.3	288.8	295.4
TPC-C	199.9	234.0	261.2
Total	449.2	522.8	556.6

Table 2: Avg. Tput. (io/sec) for E3 using the real traces.

In order to evaluate our scheme in a more realistic environment, we present results with two real traces. The first trace is obtained measuring Openmail [19] in a production environment with thousands of users. We use the TPC-C trace, explained in the previous experiment, as the second workload class, but scale down its inter-arrival times to increase its burstiness. The arrival process for the two traces are shown in Figures 7(e) and (f). Similarly to the experiment with the synthetic traces, we use loose SLOs which are $(200io/s, 600ms)$ for TPC-C and $(250io/s, 900ms)$ for Openmail. The entire trace runs for 600 seconds. In this experiment, the ADAPTIVE's parameters are set to: $P_{rt} = 20\%$, $P_{inc} = 2.5\%$, and $P_{dec} = 5\%$.

Both classes are bursty and the average aggregate rate is more than



(a) Arrival rates for two classes (b) Tput. for FIXED+AVATAR

(c) Tput. for ADAPTIVE+AVATAR (d) Tput. for SARC+AVATAR

(e) Arrival rates for Openmail (f) Arrival rates for TPC-C

(g) Aggr. Tput. for FIXED+AVATAR (h) Aggr. Tput. for ADAPTIVE+AVATAR

(i) Aggr. Tput. for SARC+AVATAR

Figure 7: Utilizing the spare bandwidth (E3). The x-axis is time (sec).

the rate the system can handle causing the system to operate close to its capacity most of the time. The stipulated SLOs are satisfied by all three schemes, i.e., the miss ratios are less than 5%. Figures 7(g), (h) and (i) show the aggregate throughput for both classes for the three schemes. One can visually see evidence of higher aggregate throughput delivered by SARC, compared to the other two rate controllers. In general, there are more spikes leading to higher throughput over the duration of the experiment, showing that there is better adaptability in utilizing spare bandwidth. Even the average throughput for the entire experiment, presented in Table 2, indicates that SARC's throughput is 10% and 5% higher than FIXED's and ADAPTIVE's throughputs, respectively. Note that the higher the throughput for each class (and not just the aggregate throughput), the better the fairness on distributing spare bandwidth.

5. CONCLUDING REMARKS

This paper has presented a new 2-level framework for meeting multi-dimensional performance virtualization goals in deploying shared storage systems. It can accommodate several workloads accessing the underlying storage utility, while meeting their individual Service Level Objectives (SLOs). The SLO is quite flexible in allowing the specification of latency and throughput constraints. At the same time, the SLO is much more stringent, and consequently more useful, than what was used in earlier studies by reducing the variability of response times.

Instead of requiring a detailed performance model, the low level scheduler of our framework - AVATAR - uses feedback to control the relative importance of deadline vs. throughput based scheduling of requests for fast adaptation to transient conditions. We have experimentally demonstrated that it can provide better adaptability to the most closely related feedback controller (Facade) proposed until now, in meeting latency constraints. In addition, the high level rate controller, SARC, is not only able to isolate the classes from each other, but can also fairly distribute any spare bandwidth to these classes towards providing better aggregate throughput.

Note that the SARC and AVATAR algorithms are not time consuming, and in fact, most of the code is not in the critical path of I/O processing. Only the insertion in the EDF queue is to some extent in the critical path, and its cost would be $O(\lg m)$ if there are m classes. One possible concern that the reader may have is that decision making in our framework is somewhat centralized, which can make it less scalable. However, we believe that performance virtualization of a large scale storage system needs hierarchical schemes, and in our ongoing work we are investigating how this 2-level framework can be used as a basic building block for a more decentralized hierarchical solution.

6. REFERENCES

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the Conference on File and Storage Technology (FAST'02)*, pages 175–188, January 2002.

[3] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *ICMCS, Vol. 2*, pages 400–405, 1999.

[4] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtulization for large-scale storage systems. In *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, October 2003.

[5] Z. Dimitrijevic and R. Rangaswami. Quality of Service Support for Real-time Storage Systems. In *Proceedings of International IPSI-2003 Conference*, October 2003.

[6] G. Ganger, B. Worthington, and Y. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*. http://www.pdl.cmu.edu/DiskSim/.

[7] J. Gemmell, H. Vin, D. Kandlur, P. Rangan, and L. Rowe. Multimedia Storage Servers: A Tutorial. *IEEE Computer*, 28(5):40–49, 1995.

[8] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, October 1996.

[9] P. Goyal, D. Jadav, D. S. Modha, and R. Tewari. CacheCOW: QoS for Storage System Caches. In *Proceedings of the 8th International Workshop on Quality of Service (IWQoS 03), Monterey, CA*, 2003.

[10] L. Huang, G. Peng, and T.-C. Chiueh. Multi-dimensional storage virtualization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[11] W. Jin, J. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[12] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of the 9th International Workshop on Quality of Service (IWQoS 04)*, 2004.

[13] B.-J. Ko, K.-W. Lee, K. Amiri, and S. Calo. Scalable service differentiation in a shared storage cache. In *23rd International Conference on Distributed Computing Systems*, May 2003.

[14] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Proceedings of the Conference on File and Storage Technology (FAST'03)*, pages 89–102, April 2003.

[15] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case. *IEEE/ACM Transactions on Networking*, pages 344–357, June 1993.

[16] Shared Storage Model. http://www.snia.org/tech_activities/shared_storage_model/.

[17] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Real Time Systems Journal: Special Issue on Flexible Scheduling of Real-Time Systems*, pages 9–47, January 2002.

[18] E. Shriver, A. Merchant, and J. Wilkes. An Analytical Behavior Model for Disk Drives with Readahead Caches and Request Reordering. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 1998.

[19] The Openmail trace. http://tesla.hpl.hp.com/private_software/.

[20] S. Uttamchandani and K. Voruganti. Polus: Growing Storage QoS Management Beyond a "4-Year Old Kid". In *Proceedings of the Conference on File and Storage Technology (FAST'04)*, March 2004.

[21] M. Uysal, G. A. Alvarez, and A. Merchant. A Modular, Analytical Throughput Model for Modern Disk Arrays. In *Proceedings of the Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2001)*, August 2001.

[22] Virtualization(I,II). http://www.snia.org/education/tutorials/.

[23] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, November 1994.

[24] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger. Storage Device Performance Prediction with CART Models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[25] WebSearch trace. http://traces.cs.umass.edu/storage/.

[26] J. Wilkes. Traveling to Rome: QoS Specifications for Automated Storage System Management. In *Proceedings of the International Workshop on Quality of Service*, June 2001.

[27] B. Worthington, G. Ganger, and Y. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1994.

[28] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83, October 1995.

[29] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet-switched Networks. *ACM Transactions on Computer Systems (TOCS)*, pages 101–124, May 1991.

[30] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Usenix Technical Conference*, June 2001.

APPENDIX

A. DETAILS ON THE ADAPTIVE APPROACH

The mechanism used by the APAPTIVE scheme of E2 to change the full credit amounts is as follows: In time window k, we denote the deadline of each class i as D_i, and the 95$^{\text{th}}$ percentile of response time of class i in the low level as $T_i(k)$. P_{rt}, P_{inc}, and P_{dec} are three parameters that indicate how each full credit amount is updated. We call the full credit amount for a class that is initially set as the original credit amount.

1. If all classes satisfy their deadlines in the low level, and $(D_i - T_i(k))/D_i > P_{rt}$, increase full credit amounts by P_{inc} for each class.
2. If at least one class (let us assume class i) can not satisfy the deadline, and $(T_i(k) - D_i)/D_i < P_{rt}$, then check if the second class (let us assume class j) can satisfy its deadline and if its full credit amount is above its original credit number. If so, decrease its full credit amount by P_{dec}; otherwise, decrease the full credit accounts of both classes by P_{dec}.
3. If $(T_i(k) - D_i)/D_i \geq P_{rt}$, then reset the full credit amounts of both classes to their original credit amounts.
4. If the full credit amount of a class is less than its original credit amount, reset it to its original credit amount.